# Exception handling sequential processes : design, verification and implementation

Jean-Pierre Banâtre, Valérie Issarny

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

1992

25ème anniversaire

N° 1710

*Programme 1*
*Architectures parallèles, Bases de données,*
*Réseaux et Systèmes distribués*

# EXCEPTION HANDLING IN COMMUNICATING SEQUENTIAL PROCESSES
## Design, Verification, and Implementation

**Jean-Pierre BANÂTRE**
**Valérie ISSARNY**

**Juin 1992**

*RR-1710*

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX FRANCE
Tél. : 99 84 71 00 - Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

# Exception Handling in Communicating Sequential Processes

## Design, Verification, and Implementation

Jean-Pierre Banâtre

Valérie Issarny*

INRIA-Rennes / IRISA

Campus de Beaulieu, 35042 Rennes Cédex, FRANCE

Publication Interne n° 660 - Mai 1992 - 38 pages

**Abstract**

The design of an exception handling mechanism for communicating sequential processes is presented. It is primarily concerned with correctness of parallel programs using the mechanism. The proposed mechanism relies on only two basic additions to the notions already needed to cope with sequential program exceptions. To demonstrate that the exception handling mechanism serves the design of correct robust parallel programs, a sound and relatively complete proof system is introduced for the enriched host programming language. Furthermore, the adequacy of the exception handling mechanism with the underlying programming model is shown. The programming language integrating the mechanism may be rewritten in terms of commands of the embedding language. Finally, comparison with related work is described.

**Key words:** CSP, exception handling, language constructs, axiomatics semantics, program transformation.

*Second author's present address: University of Washington, Dept. of Computer Science and Engineering, FR-35, Seattle, WA 98195, USA.

# Traitement des Exceptions et Processus Séquentiels Communicants

## Conception, Vérification et Mise en œuvre

**Résumé**

Dans cet article, nous proposons un mécanisme de traitement d'exceptions pour processus séquentiels communicants, dont la conception est essentiellement guidée par la volonté de simplifier l'écriture de programmes parallèles robustes corrects. Le mécanisme présenté repose uniquement sur la définition de deux nouvelles notions hormis celles déjà nécessaires dans le cadre du traitement des exceptions séquentiel. Afin de montrer que le mécanisme que nous proposons facilite la conception de programmes parallèles robustes corrects, nous introduisons un système de preuve, prouvé correct et relativement complet, pour le langage hôte enrichi de la facilité de traitement d'exceptions. Par ailleurs, nous montrons que le mécanisme de traitement d'exceptions est compatible avec le modèle de programmation considéré. Le langage de programmation intégrant le mécanisme peut être réécrit en terme de commandes du langage de base. Enfin, nous comparons notre proposition avec des travaux apparentés.

**Mots clés:** CSP, traitement des exceptions, constructions linguistiques, sémantique axiomatique, transformation de programmes.

# 1 Introduction

An important issue in programming language design is to simplify the development of correct programs. Program correctness is defined according to standard input and output assertions: when the program is executed in an initial state satisfying the standard input assertion, it returns a result satisfying the standard output assertion. Programs should be designed to behave correctly even in the presence of exceptional circumstances that are not considered in their standard input assertions. Such programs are termed *robust*, *reliable*, or *fault tolerant*. To help the design of robust programs, dedicated mechanisms, such as *exception handling mechanisms*, have been integrated within programming languages. Main features of exception handling mechanisms designed for sequential languages are now well understood. On the other hand, exception handling for parallel programs is still an evolving subject with no clear consensus. This paper addresses exception handling within parallel programs and notably focuses on the correctness issue of resulting programs.

## 1.1 Exception Handling Terminology

We present some common terminology to describe exception handling. An operation (e.g., a procedure) that is invoked in an initial state belonging to the operation's *standard domain*, a subset of the operation's domain, returns a result satisfying the operation's standard output assertion. On the other hand, if an operation is invoked in an initial state outside the operation's standard domain, this state belongs to the operation's *exceptional domain*. The actual detection of the exception during operation execution leads to exception *raising* which is followed by the execution of a specific computation, called *exception handling*. If the exception cannot be handled within the operation where it was raised, the exception is *signalled* to the embedding environment and the operation is referred to as the *exception signaller*. A *model of exception handling* then defines the interaction between a signaller and its handler. Finally, an *exception handling mechanism* defines a set of appropriate language constructs that are integrated within a programming language to express a given model of exception handling.

## 1.2 Sequential Exception Handling Models

There exist two major sequential exception handling models [Knudsen87]: the *continuation model* and the *termination model*. In the continuation model, the

signaller suspends its execution, invokes the handler and resumes its activity. In the termination model, signalling an exception causes the termination of the operation raising the exception and the subsequent execution of the handler.

The main advantage of the termination model stands in its simplicity. It introduces very few primitives in the host language, that is, a command for explicit signals and a command to define exception handling scope rules. Examples of languages using the termination model are CLU [Liskov et al.79], ADA [Ada83], MODULA2 + [Rovner et al.85] and EIFFEL [Meyer88].

On the other hand, the continuation model is intrinsically complex. In the pioneering proposal of [Goodenough75], three cases dependent upon the exception signaller are considered: the signaller has to be resumed, the signaller must not be resumed, and resumption is optional. The more recent proposal of [Yemini et al.85] alleviated this complexity. Even though this proposal introduces a model of exception handling (called the *replacement model*) as powerful as the continuation model, it requires few primitives to be added in the host programming language. However, this model still remains more complex than the termination one.

## 1.3 Parallel Exception Handling

Exception handling has been addressed for various parallel programming models including imperative programming languages with explicit parallelism. We may classify existing proposals according to three categories. Proposals for ADA [Ada83], ARGUS [Liskov et al87], NIL [Strom et al.83], ABCL-1 [Ichisugi et al.90], and SR [Huang et al.90] mainly cope with exception handling in the presence of synchronous or asynchronous remote operation call. The work of [Campbell et al.86, Taylor86] and [Jalote et al.86] addresses exception handling when parallel operations can be nested. Finally, propagation of exceptions to remote processes has been dealt with in [Szalas et al.85] and [Levin77]. These proposals were not directly concerned with correctness of robust parallel programs. Even though a proof system has been defined for a subset of ADA with exception handling [Lodaya et al.90], this work was done *a posteriori* and did not influence design choices.

In this paper, we propose a model of exception handling for parallel programs. However, parallel programming being a wide area [Bal et al.89], we restrict our discussion to exception handling within Systems of Communicating Processes (referred to as SCP in the remainder of this paper). Our main design goal is to

provide a model of exception handling for the design of correct and robust parallel programs. As a first claim, we believe that a model of exception handling for parallel programs should introduce as few control structures as possible; it should not add complexity to existing parallel control structures. We therefore choose to extend the termination model. This extension is primary concerned with deadlock avoidance. Let us consider exception occurrence within a process component of a SCP. If the process can locally handle the exception, sequential exception handling may be used. On the other hand, if the process cannot hide the exception, and assuming that the signalling process was about to communicate synchronously with a process $P$, exception occurrence prior to this communication leads to $P$'s blocking. It follows that specific control structures are required to avoid deadlock.

The remainder of this paper is organized as follows. Section 2 briefly describes the language CSP [Hoare78] used as a support for our discussion. Section 3 defines our model of exception handling together with its integration within CSP. The resulting programming language is called $\mathcal{E}_{CSP}$. Section 4 deals with the issue of the verifiability of our model. A sound and relatively complete proof system is presented for proving partial correctness of $\mathcal{E}_{CSP}$ programs. Section 5 shows that every $\mathcal{E}_{CSP}$ program may be rewritten into a CSP program. Conclusions and assessment of our proposal are offered in Section 6.

# 2 The Embedding Language

Our exception handling model is intended for communicating sequential processes, and as a reference language we have chosen CSP [Hoare78]. In the following definitions, *source* and *destination* represent process labels, $x$ a variable, and $n$ a number.

| | | | |
|---|---|---|---|
| SC | ::= | **skip** \| x := E \| S \| R | *simple commands* |
| S | ::= | destination!E | *send command* |
| R | ::= | source?x | *receive command* |
| E | ::= | n \| x \| (E) \| E+E \| E-E \| E×E \| E *div* E | *expression* |

The two last commands enable a synchronous communication according to the *rendezvous* principle. A communication occurs between two processes whenever (*i*) a *receive command* in one process specifies as its source the process name of the other process; (*ii*) a *send command* in the other process specifies as its destination the process name of the first process; and (*iii*) the variable of the receive command type-matches the value denoted by the expression of the send

command. On these conditions, the send and receive commands are said to be *matching*. Matching commands are executed simultaneously and their combined effect is to assign the value of the expression of the send command to the variable of the receive command.

We now present the composite commands. Brackets "< >" are introduced into BNF to denote zero or several repetitions of the enclosed material. B denotes the syntactic domain of boolean expressions, and D, the domain of declarations. For the sake of simplicity, we consider only integer variables.

| | | | |
|---|---|---|---|
| P | ::= | [PROCESS < \|\| PROCESS > ] | *parallel command* |
| PROCESS | ::= | processlabel :: CL | *process declaration* |
| CL | ::= | D; C <; C > | *command list* |
| D | ::= | **var** x <; D> | *declarations* |
| C | ::= | SC \| RC \| AC | *commands* |
| RC | ::= | *[ G → C < □ G → C> ] | *repetitive command* |
| AC | ::= | [ G → C < □ G → C> ] | *alternative command* |
| G | ::= | B <; S \| R $>^{0/1}$ | *guard* |
| B | ::= | E ◇ E \| **not** B \| B **or** B \| B **and** B | *boolean expressions* |
| ◇ | ::= | = \| < \| > | |

The *parallel command* expresses concurrent execution of sequential commands. Each constituent of the parallel command is called a *process*. Due to communication commands, processes must be named (*processlabel*). Alternative and repetitive commands are based on guarded commands [Dijkstra75]. A guarded command is executed only if and when the execution of its *guard* does not fail. A guard fails if its boolean expression denotes false. If the boolean expression denotes true, it has no effect and the communication command[1] at the end of the guard - if any - is executed only if and when its matching command is executed. If several guards may be selected when executing an *alternative command*, only one is chosen arbitrarily and the others have no effect. On the other hand, if none of the guards can be selected, the alternative command fails and the enclosing process terminates. When all guards of a *repetitive command* fail, the repetitive command terminates. Otherwise, its body (i.e., the alternative command) is executed once and the whole repetitive command is taken again. A repetitive command may have communication commands within guards. If all processes named in the guards have terminated, then the repetitive command also terminates.

[1] We consider an extended CSP: receive commands may appear in guards.

In order to describe several processes built according to the same model, the abbreviation P(i:1..N) :: CL represents [$P_1$ :: $CL_1$ || ... || $P_n$ :: $CL_n$]. We also use the abbreviation $*[\Box_{k=1..g}\ G_k \rightarrow C_k]$ for $*[G_1 \rightarrow C_1 \Box ... \Box G_g \rightarrow C_g]$.

# 3 Exception Handling

In this section, we define the exception handling mechanism that extends CSP. An overview of the base exception handling model is first presented.

## 3.1 Model of Exception Handling: Informal Presentation

The occurrence of an exception within a process may interfere with other processes of the SCP. As an example, let us consider two processes $P_1$ and $P_2$ belonging to the same SCP:

$P_1$ :: ...; x := *integer expression*; $\underline{P_2 \,!\, x}$; ...
$P_2$ :: ...; $\underline{P_1 \,?\, t}$; ...;

The computation of the integer expression within $P_1$ may result in signalling an exception (e.g., overflow). It follows that $P_2$ may become blocked, waiting for a communication with $P_1$. Such a deadlock is unacceptable and the termination model has to be extended. An exception whose occurrence prevents achievement of an expected rendezvous leads to the exceptional termination of its signaller. Such an exceptional termination is expressed by signalling a *global exception*, which indicates that the signaller failed to ensure an expected cooperation with at least one other process.

The exceptional termination of a process affects the behavior of processes expecting a communication with it. In the previous example, assuming that exception occurrence within $P_1$ results in signalling a global exception e, process $P_2$ becomes blocked. Therefore, the global exception e has to be *caught* and processed by $P_2$. More generally, a process *catches* a global exception only if and when it communicates synchronously with the exception signaller. When a process catches a global exception, the handler of the exception is sought within the process as in the sequential case (i.e., the exception is defined locally to the catching process).

It is important to notice that, unlike processes signalling global exceptions, a process catching a global exception is not compelled to terminate. Actually, we investigated such a possibility, which led us to integrate the resumption facility within the exception handling model [Issarny90]. A major disadvantage of this approach was that correctness proofs of parallel programs using exception handling turned out to be too complex. On the other hand, termination of a process issuing a global exception is required even though this may seem too restrictive. Let us examine the alternative solution that relies on the decomposition of a process into a sequence of possibly nested actions. We now consider that a global exception signal leads to the termination of the enclosing action instead of the whole process. A process then catches a global exception only if and when it executes a communication command that *semantically* matches a communication command belonging to the process action where the exception is signalled. This solution requires the definition of additional programming constructs to clearly identify processes actions that contain semantically matching pairs of communication commands when the SCP executes. Integrating this facility within a parallel programming language is equivalent to providing a means to declare "recoverable actions" within a SCP, that is, parallel operations that can be nested. We believe that such a construct which may be useful, has not to be defined as a part of an exception handling mechanism. Instead, it has to be introduced independently and the exception handling mechanism should be extended to cope with this new language feature. As a first result, the programming construct integrated to define nested parallel operations is not only devoted to exception handling and may serve other purposes. For example, the SCRIPT abstraction mechanism enables the low level details that implement patterns of communication to be hidden [Francez et al.86], while the notion of *multiprocedure* enables recursive parallel operations [Banatre et al.86].

As a conclusion, let us emphasize that the main function of our exception handling model amounts to properly cope with exceptional termination of a process when the process is no longer able to cooperate normally with other processes of the SCP. The remainder of this paper highlights the benefits of such a feature. The model requires addition of very few commands to the host language.

## 3.2 Syntax

We present changes to CSP syntactic domains for expressing the above model of exception handling. The introduction of global exceptions places requirements

on process declaration: a process should declare the global exceptions it signals. The syntactic domain PROCESS defining a process is thus modified so that signalled *global* exceptions may be stated in the process header[2]. Domain PROCESS becomes:

| | | | |
|---|---|---|---|
| PROCESS | ::= | processlabel **signals** EL :: CL | *process declaration* |
| EL | ::= | e < , e > | *exception list* |

where EL is the list of global exceptions signalled by the process. The commands of the exception handling mechanism, added to the syntactic domain *C*, are exactly those needed to express exception handling within sequential programs. We identify the *signal command* (SIG) that explicitly signals an exception, and the *exception handling command* (EHC) that permits attachment of a list of handler declarations to a block:

| | | | |
|---|---|---|---|
| SIG | ::= | **signal** e | *signal* |
| EHC | ::= | **try** C **except** HL **end** | *exception handling* |
| HL | ::= | EL: C <; EL: C > | *handler list* |

An exception handling command defines an exception handling scope. Its semantics is: if an exception $e$ is signalled within C then C terminates and $e$'s handler is sought within the handler list HL. However, if a handler is not declared for the signalled exception, the exception is propagated to the enclosing exception handling scope, that is, the propagation of exceptions is *implicit*. This informal semantics of exception handling defines the termination model. Nonetheless, let us recall that the exceptional terminational of a process $P_i$ due to the signal of a global exception leads to propagating the exception to processes attempting to communicate with $P_i$. In other words, global exception handling introduces horizontal propagation of exceptions in addition to the hierarchical propagation of exceptions usually encountered within sequential programs. Exception handling modifies the semantics of communication commands as defined in CSP: exception catching may replace normal communication. Hence, the action which is performed when executing a communication command can be one of the two following:

($i$) normal execution of the corresponding CSP command; or

[2] It should be noticed here that static creation of processes would enable compile time inference of global exceptions signalled by a process. However, the explicit setting of global exceptions within process header improves program readability and enforces static checking.

(*ii*) catching of the global exception signalled by the process named in the communication and execution of the appropriate handler.

Finally, in the rest of the paper, it is assumed that the evaluation of an integer expression E may result in the signal of the predefined exception *unrep*. This exception indicates that the value to be computed is not machine representable. Furthermore, the evaluation of a boolean expression is supposed to never lead to an exception occurrence.

For the sake of illustration, we rewrite the example of subsection 3.1 with exception handling. Moreover, we introduce two other processes, $P_3$ and $P_4$, synchronizing with $P_1$.

```
P1 signals e :: P4 ! z; try x := exp except unrep : signal e end; P2 ! x; P3 ! y;
P2           :: ...; try P1 ? t; ... except e: h' end; C'
P3           :: ...; try P1 ? u; ... except e: h" end; C"
P4           :: P1 ? v; ...
```

Let us assume an execution of the above SCP such that the value of the integer expression *exp* is not machine representable. Process $P_1$ thus signals the global exception $e$. Exception $e$ is propagated to $P_2$ and $P_3$ since $P_1$ is expected to communicate with them after it computes the integer expression. $P_2$ catches $e$ when it communicates with $P_1$, which leads to the execution of the handler $h'$ of $e$. Once $h'$ terminates, the standard execution of $P_2$ resumes after the exception handling command, that is, $C'$ is executed. In the same way, $P_3$ catches $e$ when it attempts to communicate with $P_1$, executes the handler $h"$, and resumes its standard execution at the beginning of $C"$. On the other hand, assuming there is only one communication between $P_1$ and $P_4$, $P_4$ does not catch $e$ since the rendezvous occurs prior to $P_1$ signalling $e$.

## 3.3 Example: Cooperating Data Servers

We conclude our informal presentation of the exception handling mechanism by an example investigating security and reliability of data management using encryption and fragmentation. Data (e.g., a file or a record) is split up into $n$ fragments, which are distributed over $n$ processes $fp(i)$. A $n+1^{th}$ process, *key*, holds a key deduced from the data. A fragment $d_i$, $1 \leq i \leq n$, of data $d$ can be computed from a key and the $n$-1 fragments $d_j$, $j \in [1,n]-\{i\}$, located on other processes $fp(j)$. This is achieved by means of a function, called $\mathcal{F}$, taking as arguments the $n$-1 fragments and a key. Finally, when a process $P$ wants to access

```
fp(i:1..n) signals unavailable ::
   var
      data: storage; frgt: fragment; ad: address;
   try
      *[manager ? read(ad) —
         try read_d(data, ad; frgt); manager ! frgt                    (1)
         except err_d: signals unavailable;                            (2)
         end
      □ Other operations — ...
      ]
   except
      error:                                                           (3)
         Global exception error, signalled by manager, is caught;
         some clean-up actions may then be specified prior to process termination.
   end
```

Figure 1: Processes $fp(i)$

a data, it sends its request to the process *manager* that collects data fragments from processes $fp(i)$.

As an example of fragmentation, we may set a fragment as being a data byte and $n = 3$. Data bytes are successively distributed on each process $fp(i)$. The *key* value is equal to an *exclusive or* performed on three bytes. The $\mathcal{F}$ function is an *exclusive or* too. If $d = 0111\ 1000\ 1101$ then $d_1 = 0111$, $d_2 = 1000$, and $d_3 = 1101$. Applying *exclusive or* on the $d_i$'s, we obtain $key(d) = 0010$. If lost, every byte $d_j$ can be restored by computing $\mathcal{F}(n-1\ fragments,\ key(d))$; for instance, $d_1 = d_2 \oplus d_3 \oplus key(d)$.

Processes $fp(i)$, $1 \leq i \leq n$, are given in figure 1. Only issues relevant to exception handling are detailed, others are written as comments in italics. Furthermore, only the read operation is examined. Operation *read_d* (line (1)) accesses a fragment located at the address *ad* within the whole data, *data*, maintained by the calling process. This operation returns the requested data fragment if available; it signals *err_d* otherwise. Assuming requested data fragment unavailability, $fp(i)$ signals the global exception *unavailable* (line (2)) since the process fails to cooperate with *manager*. Finally, $fp(i)$ may catch exception *error* (line (3)) which is signalled by *manager*; this exception will be further discussed later when pro-

```
manager signals error ::
  data: storage; terminated: array [1..n+1] of boolean initialized to false
  datum: array [1..n] of fragment; read_f: array [1..n] of boolean;
  ad: address; exc: boolean; site: integer; k: fragment;
  try
    *[P ? read(ad) →
        exc := false; Initialization of read_f[i], 1 ≤ i ≤ n, to false;
        *[□_{j=1..n} not terminated[j] and not read_f[j]; fp(j) ! read(ad) →
          try read_f[j] := true; fp(j) ? datum[j]
          except
            unavailable:                                                    (1)
              terminated[j] := true ;
              [exc — signal unrestorable □ not exc → exc := true; site := j]  (2)
          end
        □_{j=1..n} terminated[j] and not read_f[j] →
          try read_f[j] := true; read_d(data, translate(ad, j); datum[j]);
          except
            err_d: [exc → signal unrestorable □ not exc → exc := true; site := j]  (2)
          end
        ];
        [ not exc — skip
        □ exc →                                                             (3)
          [not terminated[n+1] →
            key ! get_key(ad);
            try key ? k; except unavailable: signal unrestorable end        (2)
          □ terminated[n+1] →
            try read_d(data, translate(ad, n+1); k);
            except err_d: signal unrestorable;                              (2)
            end
          ];
          datum[site] := F(datum[1], ..., datum[site-1], datum[site+1], ..., datum[n], k);
        ];
        P ! datum;
    □ Other operations — ...
    ]
  except
    unrestorable: Clean-up actions; signal error                            (4)
  end
```

Figure 2: Process *manager*

cess *manager* is introduced. Process *key* holds data keys which are requested by process *manager*, its implementation, not provided here, is similar to processes $fp(i)$'s implementation.

When a process $fp(i)$[3] terminates exceptionally, its data are maintained by the process *manager* (see fig. 2), the data of *manager* being assumed to be stored in a more reliable way than those accessed by processes $fp(i)$. Assuming that $fp(i)$ is terminated, the address *ad* of a data previously located on $fp(i)$ is now given by the operation *translate* which takes *ad* and $i$ as input parameters. Furthermore, exceptional termination of processes $fp(i)$ is memorized within the array *termination* whose $j^{th}$ element is set to *true* when $fp(j)$ is known to be terminated. Data management could be implemented in an even more reliable way; for instance, additional processes could be declared so as to replace unavailable processes. However, for the sake of conciseness, this additional feature is not considered here. Finally, data located on an exceptionally terminated process are recovered in *manager*'s data by means of a dedicated process, which is not presented here.

Process *manager* collects data fragments from processes $fp(i)$. Unavailability of a data fragment is detected through the catching of the exception *unavailable* (line (1)). If more than one fragment is unavailable then process *manager* signals the exception *unrestorable* (line (2)) whose handling leads to signalling the global exception *error* (line (4)). If only one fragment is unavailable, the whole data may be restored by means of function $\mathcal{F}$ assuming data key is still available. This restoration is achieved once *manager* has collected all available data fragments (line (3)).

# 4 Axiomatic Semantics

In this section, we propose a formal definition of our exception handling mechanism. We define proof rules for proving partial correctness assertions about $\mathcal{E}_{CSP}$ programs. Verification systems based on Hoare's logic [Hoare69] have been proposed for a wide variety of programming constructs including exception handling mechanisms. For instance, mechanisms expressing the continuation model have been formalized in [Levin77, Cocco et al.82, Yemini et al.87, Cocco et al.82] and [Szczepanska91], while mechanisms expressing the termination model have

[3]The same method is applied for process *key* which is considered as process $fp(n+1)$ by process *manager*.

been in [Luckham et al.80, Cristian84] and [Lodaya et al.90]. However, in most of the above references, the class of programs for which the presented verification system is complete is not determined. To our knowledge, only the work of [Cristian84, Lodaya et al.90] and [Szczepanska91] clearly addresses this issue. Furthermore, exception handling in the presence of parallelism is only grasped in [Lodaya et al.90] where a verification system for a subset of ADA [Ada83] is defined. Finally, although exception handling mechanisms proposed in [Levin77] and [Luckham et al.80] cope with parallel issues, the given axiomatic definitions are limited to the sequential case.

## 4.1 Background

Prior to the introduction of a Hoare-like verification system for $\mathcal{E}_{CSP}$ programs, we recall some basic grounds.

Axiomatic definition of the CSP programming language gave rise to many proposals [Apt et al.91]. Here, we consider the verification system $\mathcal{T}$, introduced in [Apt83]. The choice of $\mathcal{T}$ is primarily due to the fact that it has been proven sound and relatively complete. Actually, the system $\mathcal{T}$ directly follows from the proposal of [Apt et al.80] except that it copes with a smaller subset of CSP described hereafter. The system $\mathcal{T}$ enables proving partial correctness of CSP programs. Therefore, any command of the form $[\Box_{j=1..m}\ b_j;\ \alpha_j \rightarrow C_j]$ is semantically equivalent to $[\Box_{j=1..m}\ b_j \rightarrow \alpha_j;\ C_j]$ where $\alpha_j$ denotes a communication command. Furthermore, the distributed termination convention of the CSP repetitive command is not handled in [Apt et al.80]. It follows that any command of the form $*[\Box_{j=1..m}\ b_j;\ \alpha_j \rightarrow C_j]$ is semantically equivalent to $*[\Box_{j=1..m}\ b_j \rightarrow \alpha_j;\ C_j]$. A consequence of these equivalences is that the proof rules dealing with the I/O guarded commands can be derived from the proof rules dealing with the equivalent constructs. In the system $\mathcal{T}$, a proof of pre- and post-assertion about a parallel program is done in two stages:

(1) separate proofs are constructed in isolation for each program component, assumptions are made locally on the properties of values exchanged;

(2) properties of complete program are deduced by composing the proofs of the component processes, previous assumptions are confirmed or denied at this stage by means of cooperation rules.

Due to the above principle, axioms and rules should be provided for every command that can be used within a process. In particular, the axiom associated with

a receive command is defined as follows:

$$p \{ P_i ? x \} q$$

This axiom enables deduction of any post-assertion $q$. However, any chosen post-assertion will be checked for correctness according to the value sent by the matching send command.

In order to cope with exception handling within $\mathcal{T}$, we retain the notation introduced in [Cristian84]. It relies on the fact that integrating an exception handling mechanism within a programming language makes most commands one-entry/multiple-exit. Compared with other existing proposals, the advantage of the undertaken approach is a clear separation between the validation of program properties in the presence and in the absence of exceptions. The proposed proof system enables proving total correctness of sequential programs. A correctness formula of the system is of the form $p \{ C \} a : q$ whose interpretation is *if C is invoked in a state satisfying p then C terminates at the exit point a in a state satisfying q*; an exit point can be either *end* or an exception label. When $a$ is equal to *end*, the abbreviation $p \{ C \} q$ is meant for $p \{ C \}$ *end*: $q$. A program $P$ with $k$ exit points is termed *robust* if it terminates at a declared exit point for any possible input state. The notion of correctness for a one-entry/multiple-exit program is a straightforward extension of that for one-entry/one-exit programs. To specify the expected behavior of a program $P$ which signals $k$ exceptions, $(k+1)$ pairs $(r_i, s_i)$, $0 \leq i \leq k$ which give pre- and post-assertions are needed [Cristian82]. The pair $(r_0, s_0)$ denotes the expected standard behavior of $P$ while pairs $(r_j, s_j)$, $1 \leq j \leq k$ specify its expected exceptional behavior. A program $P$ is partially correct if the statement $r_i \{P\} x_i : s_i$ is verified for all pairs.

The above system takes into account exceptions signalled within a sequential process. We have moreover to deal with global exception catching. This is addressed in the axioms and rules for communication commands.

## 4.2 Proof Rules

We now introduce the proof system $\mathcal{T}_{\mathcal{E}}$ which enables proving partial correctness of $\mathcal{E}_{CSP}$ programs. This presentation uses the following notations. Let $L$ denote a first-order language with equality. Its formulas are called assertions and are denoted by the letters $p$, $q$, $r$ and $s$. The simple variables are denoted by the letters $u$, $v$, $x$, $y$, $z$; the expressions by the letter $t$; the boolean expressions by the letter $b$ and the exit points by the letters $a$ and $e$. Finally, $p[t/x]$ stands for

the result of substituting $t$ for the free occurrences of $x$ in $p$. The programs that are considered are written in a subset of $\mathcal{E}_{CSP}$ whose expressions are built from non logical symbols of $L$. Processes and commands are denoted by the letter $C$. Finally, the set of the global exceptions signalled by process $P_i$ is denoted by $\mathcal{G}_i$.

The reasoning about $\mathcal{E}_{CSP}$ programs is achieved in two stages: processes are proven partially correct in isolation and then properties of the $\mathcal{E}_{CSP}$ program are deduced. A correctness formula of the system $\mathcal{T}_\mathcal{E}$ is of the form:

$$p \{ \text{ C } \} a\text{: } q$$

when $C$ is a sequential $\mathcal{E}_{CSP}$ command. The informal meaning of of $p \{ \text{ C } \} a\text{: } q$ is:

> *if $p$ is true before C's execution and if C terminates then, after C's execution, the exit point is $a$ and $q$ is true.*

By convention, we omit stating the exit point when it is equal to *end*. When a correctness formula states the property of an $\mathcal{E}_{CSP}$ parallel program, it is of the form:

$$p \{ \text{ C } \} (a_1, ..., a_n)\text{: } q$$

where $n$ is the number of C's processes, and $a_i$'s are exit points. The informal meaning of this last formula is:

> *if $p$ is true before* C*'s execution and if* C *terminates, then, after C's execution, the exit point of the $i^{th}$ $(1 \leq i \leq n)$ process is $a_i$ and $q$ is true.*

By convention, the list of exit points is omitted when all the exit points are equal to *end*.

### 4.2.1 Proving properties of a process $\mathbf{P_i}$

In this paragraph, commands are assumed to belong to a process $P_i$. Hence, by definition, these commands are sequential. The proof rule for the **skip** command is defined as usual:

$\mathbf{A_\mathcal{E}\ 1}$ $\quad \vdash \quad p \{ \textbf{skip} \} p$

Let us now give proof rules for an assignment command. The evaluation of an expression may result in the signal of the exception *unrep*, which indicates that the resulting integer is not machine representable. As in [Cristian84], we define this exception condition through boolean functions that take an expression as argument. The function *ovf* returns *true* if the evaluation of its argument leads to overflow, function *udf* being applied to determine underflow occurrence. Finally, the function *repr* returns *true* if the value of its argument is machine representable. These functions, not detailed here, are all three defined by induction on the structure of expressions. Two proof rules define an assignment depending upon whether its right-hand side is machine representable or not. In the first case, the command terminates normally while, in the second case, the exit point becomes *unrep*:

$$\mathbf{R}_{\mathcal{E}}\ \mathbf{1} \quad \left| \quad \frac{p \Rightarrow representable(t)}{p[t/x]\ \{\ \mathrm{x} := \mathrm{t}\ \}\ p} \right.$$

$$\mathbf{R}_{\mathcal{E}}\ \mathbf{2} \quad \left| \quad \frac{p \Rightarrow (ovf(t) \vee udf(t))}{p\ \{\ \mathrm{x} := \mathrm{t}\ \}\ unrep : p} \right.$$

The semantics of $\mathcal{E}_{CSP}$ communication commands differs from that of corresponding CSP commands; a global exception may be caught. Furthermore, the evaluation of the expression specified in a send command may lead to the signal of the exception *unrep*.

$$\mathbf{R}_{\mathcal{E}}\ \mathbf{3} \quad \left| \quad \frac{p \Rightarrow representable(t)}{p\ \{\ \mathrm{P}_j\ !\ \mathrm{t}\ \}\ a : q} \right.$$

The above rule implies that *any* post-assertion $q$ as well as *any* exit point $a$ can be deduced after a send command. Note, however, that $q$ and $a$ cannot be arbitrary since they must pass the cooperation test at the second stage of the proof.

$$\mathbf{R}_{\mathcal{E}}\ \mathbf{4} \quad \left| \quad \frac{p \Rightarrow (ovf(t) \vee udf(t))}{p\ \{\ \mathrm{P}_j\ !\ \mathrm{t}\ \}\ unrep : p} \right.$$

Finally, the receive command is defined by an axiom:

$$\mathbf{A}_{\mathcal{E}}\ \mathbf{2} \quad \left| \quad p\ \{\ \mathrm{P}_j\ ?\ \mathrm{x}\ \}\ a : q \right.$$

Axioms and rules defining commands of the exception handling mechanism are similar to those given in [Cristian84] for corresponding commands. When a command **signal** is encountered, the specified exception, *global or not*, becomes the exit point of the enclosing exception handling command:

$\mathbf{A_{\mathcal{E}}\ 3}$ | $p\ \{\ \mathbf{signal}\ e\ \}\ e : p$

When the exit point, $a$, of a command enclosed in an exception handling command does not belong to the list of handled exceptions, the exit point of the exception handling command is $a$. Let us consider the following exception handling command: **try** C **except** $e_1$: $C_1$; ...; $e_m$: $C_m$ **end** and $a$ be different of $e_i$ for any $i$ of $[1, m]$, we get:

$$\mathbf{R_{\mathcal{E}}\ 5} \quad \frac{p\ \{\ C\ \}\ a : q}{p\ \{\ \mathbf{try}\ C\ \mathbf{except}\ e_1 : C_1\ ;\ \ldots\ ;\ e_m : C_m\ \mathbf{end}\ \}\ a : q}$$

On the other hand, when the exit point of $C$ belongs to the list of handled exceptions, the enclosing exception handling command inherits the exit point of the exception handler:

$$\mathbf{R_{\mathcal{E}}\ 6} \quad \frac{p\ \{\ C\ \}\ e_i : q, q\ \{\ C_i\ \}\ a : r}{p\ \{\ \mathbf{try}\ C\ \mathbf{except}\ e_1 : C_1\ ;\ \ldots\ ;\ e_m : C_m\ \mathbf{end}\ \}\ a : r}$$

If the exit point, $e$, of a sequentially composed command is exceptional, the following commands are not executed and the exit point of the whole sequential composition is $e$:

$$\mathbf{R_{\mathcal{E}}\ 7} \quad \frac{p\ \{\ C_1\ \}\ e : q}{p\ \{\ C_1\ ;\ C_2\ \}\ e : q}$$

When the exit point of the first component of a sequential composition command is *end*, the whole command inherits the exit point of its second component:

$$\mathbf{R_{\mathcal{E}}\ 8} \quad \frac{p\ \{\ C_1\ \}\ q, q\ \{\ C_2\ \}\ a : r}{p\ \{\ C_1\ ;\ C_2\ \}\ a : r}$$

Let us recall that our concern is to prove *partial* correctness of programs. Therefore, as in [Apt83], communication commands within guards are discarded. An alternative command inherits the exit points of all of its constituent guarded commands that may be selected. Let $c_p$ be the set of indices such that $i \in c_p$ if and only if $p \Rightarrow b_i$, we define:

$$\mathbf{R_{\mathcal{E}}\ 9} \quad \frac{(p \wedge b_j\ \{\ C_j\ \}\ a : q)_{j \in c_p}}{p\ \{\ [\Box_{j=1,\ldots,m}\ b_j \rightarrow C_j]\ \}\ a : q}$$

Two rules define the repetitive command: one is related to standard termination and the other to exceptional termination. These rules use an invariant $p$, a variant integer expression, $t$, setting the number of iterations to be performed, and an integer $t'$.

[4] One may have noticed that, due to the rules $\mathbf{R_{\mathcal{E}}\ 10}$ and $\mathbf{R_{\mathcal{E}}\ 11}$, $\mathcal{T}_{\mathcal{E}}$ actually enables proving *weak total correctness* of $\mathcal{E}_{CSP}$ programs.

$$\mathbf{R_\mathcal{E}\ 10}\quad \frac{(p \wedge b_j \wedge (U \leq 0) \Rightarrow \neg(\wedge_{k=1}^{m} b_k))_{j=1,\ldots,m},\ (p \wedge b_j \wedge (U \leq t)\ \{\ C_j\ \}\ p \wedge (U < t))_{j=1,\ldots,m}}{p\ \{\ *[\Box_{j=1,\ldots,m}\ b_j \rightarrow C_j]\ \}\ p \wedge \wedge_{k=1}^{m} \neg b_k}$$

$$\mathbf{R_\mathcal{E}\ 11}\quad \frac{(p \wedge b_j \wedge (U \leq 0)\ \{\ C_j\ \}\ e : q)_{j=1,\ldots,m},\ (p \wedge b_j \wedge (0 < U) \wedge (U \leq t)\ \{\ C_j\ \}\ p \wedge (U < t))_{j=1,\ldots,m}}{p\ \{\ *[\Box_{j=1,\ldots,m}\ b_j \rightarrow C_j]\ \}\ e : q}$$

The following well known rules $\mathbf{R_\mathcal{E}\ 12}$ of consequence, $\mathbf{R_\mathcal{E}\ 13}$ of conjunction, and $\mathbf{R_\mathcal{E}\ 14}$ of disjunction are also used.

$$\mathbf{R_\mathcal{E}\ 12}\quad \frac{p \Rightarrow q,\ q\ \{\ C\ \}\ a : r,\ r \Rightarrow s}{p\ \{\ C\ \}\ a : s}$$

$$\mathbf{R_\mathcal{E}\ 13}\quad \frac{p\ \{\ C\ \}\ a : q,\ p\ \{\ C\ \}\ a : r}{p\ \{\ C\ \}\ a : q \wedge r}$$

$$\mathbf{R_\mathcal{E}\ 14}\quad \frac{p\ \{\ C\ \}\ a : r,\ q\ \{\ C\ \}\ a : r}{p \vee q\ \{\ C\ \}\ a : r}$$

Using the axioms and rules introduced so far, separate proofs for each process can be established. This is presented, as in [Owicki et al.76], by a *proof outline* in which each sub-statement of a process is preceded and followed by a corresponding assertion. In this proof outline, a process guesses the global exception it may catch after a communication command and the value it may receive. When the proofs are combined, those assumptions have to be checked for consistency using a cooperation test. This issue is addressed in the following paragraph.

### 4.2.2 Proving properties of parallel programs

A *global invariant* $I$ is associated with every $\mathcal{E}_{CSP}$ program. A global invariant $I$ expresses global information about its program. In particular, it helps to determine communication commands that *semantically* match within the program, that is, pairs of communication commands that actually result in communications when the program executes. Furthermore, program sections $C$ within which $I$ needs not hold are bracketed as $(C)$. The notion of *bracketing* for an $\mathcal{E}_{CSP}$ process $P_i$ is defined as follows:

**Definition 1 (Bracketing)** *A process $P_i$ is* bracketed *if the brackets "(" and ")" are interspersed in its text so that:*

*(i) for every program section* $\langle C \rangle$, *called* bracketed section, $C$ *is of the form* $C_1$; $\alpha$; $C_2$ *where* $\alpha$ *is a communication command, and* $C_1$ *and* $C_2$ *do not contain any communication command;*

*(ii) when the correctness formula associated to a bracketed section* $C$ *is of the form* $p$ { $C$ } $e$: $q$ *with* $e \neq end$, *then* $C$ *is a communication command;*

*(iii) all the communication commands that are executed, (i.e., whose execution is not preceded by an unhandled exception signal) and whose execution does not lead to the signal of the exception* unrep, *are bracketed.*

The requirement for cooperation of separate proofs is stated in the proof rule for parallel commands:

$$\mathbf{R}_{\mathcal{E}}\ \mathbf{15} \quad \frac{\text{Proofs of } p_i \{ C_i \} a_i : q_i,\ i = 1, ..., n \text{ cooperate}}{\wedge_{i=1}^{n} p_i \wedge I \{ C_1 \| ... \| C_n \} (a_1, ..., a_n) : \wedge_{i=1}^{n} q_i \wedge I}$$

provided no free variable in $I$ is subject to change outside a bracketed section. Let "$" stand for either "?" or "!" and say that two bracketed sections match if they contain matching communication commands, we define:

**Definition 2 (Cooperation)** *The proofs of the* $p_i \{C_i\} a_i$: $q_i$, $i = 1, ..., n$ *cooperate if:*

*(i) the assertions used in the proof of* $p_i \{C_i\} a_i$: $q_i$ *have no free variable subject to change in* $C_j$ $(i \neq j)$;

*(ii)* $pre(C_1) \wedge pre(C_2) \wedge I \{C_1 || C_2\} post(C_1) \wedge post(C_2) \wedge I$ *holds for all matching pairs of bracketed sections* $\langle C_1 \rangle$ *and* $\langle C_2 \rangle$ *whose enclosing correctness formula are respectively of the form* $pre(C_1) \{C_1\} post(C_1)$ *and* $pre(C_2) \{C_2\} post(C_2)$;

*(iii) for every correctness formula of the form* p {P$_i$ $ t} e: q *associated to a bracketed section* $\langle P, \$t \rangle$, *within a proof outline of a process* $P_j$, *we have:* $p \Rightarrow q$ *and* $a_i = e$ *with* $e \in \mathcal{G}_i$.

Points $(i)$ and $(ii)$ directly follow from the corresponding definition of cooperation given in [Apt83]. Point $(ii)$ copes with actual communications between processes. Point $(iii)$ deals with communications that involve a process having signalled a global exception. The following axioms and proof rules are needed to establish cooperation: the communication axiom $\mathbf{A}_{\mathcal{E}}$ **4**, the formation rule $\mathbf{R}_{\mathcal{E}}$ **16**, the preservation axiom $\mathbf{A}_{\mathcal{E}}$ **5** the substitution rule $\mathbf{R}_{\mathcal{E}}$ **17**, and the rule $\mathbf{R}_{\mathcal{E}}$ **18** for auxiliary variables.

$\mathbf{A_\mathcal{E}\ 4}$ | $\mathbf{true}\ \{\ P_i\ ?\ x \parallel P_j\ !\ t\ \}\ x = t$

provided $P_i\ ?\ x$ and $P_j\ !\ t$ are taken from $P_j$ and $P_i$, respectively.

$$\mathbf{R_\mathcal{E}\ 16} \left| \frac{p\ \{\ C_1;\ C_3\ \}\ q,\ q\ \{\ \alpha \| \bar{\alpha}\ \}\ r,\ r\ \{\ C_2;\ C_4\ \}\ s}{p\ \{\ (C_1;\ \alpha;\ C_2) \parallel (C_3;\ \bar{\alpha};\ C_4)\ \}\ s} \right.$$

provided $\alpha$ and $\bar{\alpha}$ match, $C_1$, $C_2$, $C_3$ and $C_4$ do not contain any communication commands, and no variable in $C_1$; $C_2$ is subject to change in $C_3$; $C_4$ and *vice versa*.

$\mathbf{A_\mathcal{E}\ 5}$ | $p\ \{\ C\ \}\ p$

provided no free variable of $p$ is subject to change in $C$ and the execution of $C$ in a state satisfying $p$ does not lead to an exception signal.

$$\mathbf{R_\mathcal{E}\ 17} \left| \frac{p\ \{\ C\ \}\ a : q}{p[t/z]\ \{\ C\ \}\ a : q} \right.$$

provided $z$ does not appear free in $C$ and $q$. Let $AV$ be a set of variables such that $x \in AV$ implies that $x$ appears in $C'$ only in assignments $y := t$, where $y \in AV$. Then, if $q$ does not contain free any variables from $AV$ and $C$ is obtained from $C'$ by deleting all assignments to variables in $AV$,

$$\mathbf{R_\mathcal{E}\ 18} \left| \frac{p\ \{\ C'\ \}\ a : q}{p\ \{\ C\ \}\ a : q} \right.$$

This concludes the presentation of the system $\mathcal{T}_\mathcal{E}$ whose proofs of soundness and relative completeness may be found in [Issarny91b].

## 4.3 Example: Computing Factorial

In order to illustrate the use of $\mathcal{T}_\mathcal{E}$, we develop a simple proof. This contrived example (i.e., computation of $n$ factorial) enables to detail proof of cooperation in the presence of exceptional termination of processes. The proposed implementation follows from [Banatre90]. Multiplication being associative, the system of communicating processes may be organized as a balanced binary tree. A process is associated to each node and computes the product of an integer sequence sent by its ancestor. Each process is assigned a number according to the convention used for binary trees: root has number 1 and for every node $i$, left child has number $2i$ and right child, number $2i + 1$.

```
N(1..k) ::
  var m_i; var M_i; var x_i; var y_i; var z_i;
  [N(i div 2) ? m_i →                                          (1)
     N(i div 2) ? M_i;                                         (2)
     [m_i ≠ M_i →
          x_i := (M_i - m_i + 1) div 2;
          N(2 × i) ! m_i; N(2 × i) ! (m_i + x_i - 1);          (3)
          N(2 × i + 1) ! (m_i + x_i); N(2 × i + 1) ! M_i;      (4)
          N(2 × i) ? y_i; N(2 × i + 1) ? z_i;
          N(i div 2) ! (y_i × z_i)                             (5)
     □ m_i = M_i — N(i div 2) ! m_i                            (6)
     ]
  ]
```

Figure 3: Processes N(i)

Let us first examine each process $N(i : 1..k)$ in the absence of exceptions (see fig. 3). Each process, say $N_i$, receives the lower bound $m_i$ (line (1)) and the upper bound $M_i$ (line (2)) of an ordered sequence from its ancestor, the root receiving this sequence from a user process. $N_i$ then computes the product of sequence elements. If the sequence cardinality is 1, $N_i$ returns the sequence element to its ancestor (line (6)). On the other hand, if the sequence has more than one element, $N_i$ sends the first half of the sequence to its left child (line (3)) and the second half to its right child (line (4)). After having received products $y_i$ and $z_i$ from its left and right child respectively, $N_i$ sends the product $y_i \times z_i$ to its ancestor (line (5)).

At least one exception may occur when process $N(i)$, $1 \leq i \leq k$, executes. Product evaluation may lead to the occurrence of the predefined exception *unrep*. This exception occurrence within $N(i)$ results in signalling the global exception $ov_i$. Exception $ov_i$ is caught by the signaller ancestor $N(i \ div \ 2)$ which then signals the global exception $ov_{i \ div \ 2}$. This exception is further propagated to the process ancestor which catches and propagates the exception in turn[5]. Exception propagation stops once the user process is reached. The resulting text of processes

[5] Let us remark that if the signalling process $N(i)$ is a left child, its ancestor propagates the exception to the right child $N(i \times 2 + 1)$. Nonetheless, this last process will not catch the exception since it also signals a global exception ($\prod_{i=j}^{j+k-1} i < \prod_{i=j+k}^{n} i$ where $k = (n - j + 1) \ div \ 2$).

```
N(i:2..k) signals ov_i ::
  var m_i; var M_i; var x_i; var y_i; var z_i;
  try
    [N(i div 2) ? m_i →
      N(i div 2) ? M_i;
      [m_i ≠ M_i →
          x_i := (M_i - m_i + 1) div 2;
          N(2 × i) ! m_i; N(2 × i) ! (m_i + x_i - 1);
          N(2 × i + 1) ! (m_i + x_i); N(2 × i + 1) ! M_i;
          N(2 × i) ? y_i; N(2 × i + 1) ? z_i;
          N(i div 2) ! (y_i × z_i)                          (1)
      □ m_i = M_i → N(i div 2) ! m_i ] ]
  except
    unrep, ov_2×i, ov_2×i+1: signals ov_i
  end
```

Figure 4: Processes N(i) with exceptions

$N(i)$ when $i \geq 2$ is given in figure 4. The text of $N(1)$ can easily be deduced from this presentation. Each process $N(i)$ handles global exceptions that are signalled by its children. A process also handles the exception *unrep* which results from overflow during product computation (line (1)). All the exceptions are handled in the same way: the global exception $ov_i$ is signalled in order to propagate the non computation of factorial to user.

We now consider the correctness proof of the above program fragment. We rewrite processes $N(i)$, $1 \leq i \leq k$ according to the equivalence of $[\Box_{j=1..m}\ b_j;\ \alpha_j \rightarrow C_j]$ with $[\Box_{j=1..m}\ b_j \rightarrow \alpha_j;\ C_j]$ where $\alpha_j$ is a communication command. Furthermore, the auxiliary variable $l_i$ is introduced to express semantic matching of communication commands. Proof outlines of bracketed processes $N(i)$, $2 \leq i \leq k$ are given hereafter. In these proof outlines, the following notations are used:

- pre- and post-assertions are written within braces;
- $\mathcal{M}_{/\!/}$ denotes the set of machine representable integers;
- $\beta$ denotes the integer expression $(m_i + M_i + 1)\ div\ 2$;
- $prod(a, b)$ denotes the integer expression $\Pi_{j=a}^{b} j$.
- $C$ denotes the following process fragment:

$x_i$ := ($m_i$ + $M_i$ + 1) *div* 2;
⟨ N(2 × i) ! $m_i$ ⟩; ⟨ N(2 × i) ! ($m_i$ + $x_i$ -1) ⟩;
⟨ N(2 × i + 1) ! ($m_i$ + $x_i$) ⟩; ⟨ N(2 × i + 1) ! $M_i$ ⟩;
⟨ N(2 × i) ? $y_i$; $l_i$ := $l_i$ + 1 ⟩;

The proof outline, $R_1$, in the standard case is:

{ $l_i = 0$ } ⟨ N(i *div* 2) ? $m_i$ ⟩; ⟨ N(i *div* 2) ? $M_i$; $l_i$ := $l_i$ + 1 ⟩;
{ $l_i = 1 \wedge (m_i = M_i \vee m_i \neq M_i) \wedge prod(m_i, M_i) \in \mathcal{M}_{RI}$ }
  { $l_i = 1 \wedge m_i \neq M_i \wedge prod(m_i, M_i) \in \mathcal{M}_{RI}$ }
    *C*
    { $l_i = 2 \wedge m_i \neq M_i \wedge prod(m_i, M_i) \in \mathcal{M}_{RI}$ }
    ⟨ N(2 × i + 1 ) ? $z_i$; $l_i$ := $l_i$ + 1 ⟩;
    { $l_i = 3 \wedge m_i \neq M_i \wedge prod(m_i, M_i) \in \mathcal{M}_{RI}$ }
    ⟨ N(i *div* 2) ! $y_i$ × $z_i$ ⟩;
  { $m_i = M_i \wedge prod(m_i, M_i) \in \mathcal{M}_{RI}$ }
    N(i *div* 2) ! $m_i$
{ $l_i \geq 1 \wedge (m_i = M_i \vee m_i \neq M_i) \wedge prod(m_i, M_i) \in \mathcal{M}_{RI}$ }

We set:

$$\mathcal{P}_1(i) \equiv l_i = 1 \wedge m_i \neq M_i \wedge prod(m_i, m_i + \beta - 1) \notin \mathcal{M}_{RI};$$

the proof outline, $R_2$, when the left child signals a global exception is:

{ $l_i = 0$ } ⟨ N(i *div* 2) ? $m_i$ ⟩; ⟨ N(i *div* 2) ? $M_i$; $l_i$ := $l_i$ + 1 ⟩;
{ $\mathcal{P}_1(i)$ }
  $x_i$ := ($m_i$ + $M_i$ + 1) *div* 2;
  ⟨ N(2 × i) ! $m_i$ ⟩; ⟨ N(2 × i) ! ($m_i$ + $x_i$ -1) ⟩;
  ⟨ N(2 × i + 1) ! ($m_i$ + $x_i$) ⟩; ⟨ N(2 × i + 1) ! $M_i$ ⟩;
  ⟨ N(2 × i) ? $y_i$ ⟩;
  { $ov_{2\times i}$: $\mathcal{P}_1(i)$ }
  N(2 × i + 1) ? $z_i$; N(i *div* 2) ! $y_i$ × $z_i$;
{ $ov_{2\times i}$: $\mathcal{P}_1(i)$ } **signals** $ov_i$
{ $ov_i$: $\mathcal{P}_1(i)$ }

We set:

$$\mathcal{P}_2(i) \equiv m_i \neq M_i \wedge prod(m_i, m_i + \beta - 1) \in \mathcal{M}_{RI} \wedge prod(m_i + \beta, M_i) \notin \mathcal{M}_{RI};$$

the proof outline, $R_3$, when the right child signals a global exception is:

{ $l_i = 0$ } ⟨ N(i *div* 2) ? $m_i$ ⟩; ⟨ N(i *div* 2) ? $M_i$; $l_i$ := $l_i$ + 1 ⟩; { $l_i = 1 \wedge \mathcal{P}_2(i)$ }
  *C*
{ $l_i = 2 \wedge \mathcal{P}_2(i)$ }
  ⟨ N(2 × i + 1 ) ? $z_i$ ⟩;
  { $ov_{2\times i+1}$: $l_i = 2 \wedge \mathcal{P}_2(i)$ } N(i *div* 2) ! $y_i$ × $z_i$;
{ $ov_{2\times i+1}$: $l_i = 2 \wedge \mathcal{P}_2(i)$ } **signals** $ov_i$
{ $ov_i$: $l_i = 2 \wedge \mathcal{P}_2(i)$ }

We set:

$$\begin{aligned}\mathcal{P}_3(i) \equiv\ & m_i \neq M_i \wedge \\ & prod(m_i, m_i + \beta - 1) \in \mathcal{M}_{RI} \wedge \\ & prod(m_i + \beta, M_i) \in \mathcal{M}_{RI} \wedge \\ & prod(m_i, M_i) \notin \mathcal{M}_{RI};\end{aligned}$$

the proof outline, $R_4$, when the two children send back their result but the resulting product is not machine representable is:

$\{\ l_i = 0\ \}$ ⟨ N(i *div* 2) ? $m_i$ ⟩; ⟨ N(i *div* 2) ? $M_i$; $l_i := l_i + 1$ ⟩; $\{\ l_i = 1 \wedge \mathcal{P}_3(i)\ \}$
$\mathcal{C}$
$\{\ l_i = 2 \wedge \mathcal{P}_3(i)\ \}$ ⟨ N(2 × i + 1 ) ? $z_i$; $l_i := l_i + 1$ ⟩;
$\{\ l_i = 3 \wedge \mathcal{P}_3(i)\ \}$ N(i *div* 2) ! $y_i \times z_i$;
$\{$ *unrep*: $l_i = 3 \wedge \mathcal{P}_3(i)\ \}$ **signals** $ov_i$
$\{$ *or*: $l_i = 3 \wedge \mathcal{P}_3(i)\ \}$

Now, we examine cooperation of proofs when:

- the product computed by the process $N(p)$ is not machine representable;
- the product computed by any process which is at the same depth as $N(p)$ and at the right of $N(p)$ in the process tree, is not machine representable;
- the product computed by any process which is neither at the same depth as $N(p)$ nor $N(p)$'s ancestor, is machine representable.

We use the following notations:

- the predicate $dr(p, j)$ denotes true if $N(j)$ is at the same depth as $N(p)$ and at the right of $N(p)$ in the process tree;
- the predicate $anc(p, j)$ denotes true if $N(j)$ is an ancestor of $N(p)$;
- the predicate $even(i)$ (resp. $odd(i)$) denotes true if $i$ is an even (resp. odd) number.

Let $k$ be the number of processes required to compute $n!$, the global invariant $I$ can be written as:

$$
\begin{array}{ll}
 & m_1 = 1 \wedge M_1 = n \\
\wedge & prod(m_p, M_p) \notin \mathcal{M}_{RI} \\
\wedge & \forall j : 2 \leq j \leq k, (j \neq p \wedge \neg anc(p,j) \wedge \neg dr(p,j)) \Rightarrow prod(m_j, M_j) \in \mathcal{M}_{RI} \\
\wedge & \forall j : 1 \leq j \leq k, anc(p,j) \Rightarrow prod(m_j, M_j) \notin \mathcal{M}_{RI} \\
\wedge & \forall j : 2 \leq j \leq k, dr(p,j) \Rightarrow prod(m_j, M_j) \notin \mathcal{M}_{RI} \\
\wedge & \wedge_{j=2}^{k} ((l_j \geq 1 \wedge even(j)) \Rightarrow (m_j = m_{j\ div\ 2} \wedge M_j = m_{j\ div\ 2} + x_{j\ div\ 2} - 1)) \\
\wedge & \wedge_{j=2}^{k} ((l_j \geq 1 \wedge odd(j)) \Rightarrow (m_j = m_{j\ div\ 2} + x_{j\ div\ 2} \wedge M_j = M_{j\ div\ 2})) \\
\wedge & \wedge_{j=2}^{k} (l_j \geq 2 \Rightarrow y_j = prod(m_{j\times 2}, M_{j\times 2})) \\
\wedge & \wedge_{j=2}^{k} (l_j \geq 3 \Rightarrow z_j = prod(m_{j\times 2+1}, M_{j\times 2+1}))
\end{array}
$$

We detail proof of cooperation for processes whose number is $p\ div\ 2$, $p$, $p \times 2$ and $p \times 2 + 1$ where $p$ is an even number. Cooperation of assertions for all other processes is not shown here. The following proof outlines are used:

- $R_2$ for $N(p\ div\ 2)$
- $R_4$ for $N(p)$
- $R_1$ for $N(p \times 2)$ and $N(p \times 2 + 1)$ where $N(p \times 2)$ and $N(p \times 2 + 1)$ are assumed not to be leaves.

Let us remark that points $(i)$ and $(iii)$ of **definition 2** are satisfied. Considering point $(ii)$, let us first examine pairs of semantically matching communication commands between $N(p)$ and $N(p\ div\ 2)$. According to the communication axiom $\mathbf{A}_{\mathcal{E}}$ **4**, we have:

$$
\begin{array}{c}
\mathcal{P}_1(p\ div\ 2) \wedge l_p = 0 \wedge I \\
\{\ \mathrm{N(p)\ !\ m}_{p\ div\ 2} \ ||\ \mathrm{N(p}\ div\ 2)\ ?\ \mathrm{m}_p\ \} \\
\mathcal{P}_1(p\ div\ 2) \wedge l_p = 0 \wedge I
\end{array}
$$

Using the communication axiom $\mathbf{A}_{\mathcal{E}}$ **4**, the assignment rule $\mathbf{R}_{\mathcal{E}}$ **1** and the formation rule $\mathbf{R}_{\mathcal{E}}$ **16**, we get:

$$
\begin{array}{c}
\mathcal{P}_1(p\ div\ 2) \wedge l_p = 0 \wedge I \\
\{\ \mathrm{N(p)\ !\ m}_{p\ div\ 2} \ ||\ \mathrm{N(p}\ div\ 2)\ ?\ \mathrm{m}_p;\ \mathrm{l}_p := \mathrm{l}_p + 1\} \\
\mathcal{P}_1(p\ div\ 2) \wedge l_p = 1 \wedge I
\end{array}
$$

Let us now examine pairs of semantically matching communication commands within $N(p)$ and $N(p \times 2)$. Let us consider the emission to $N(p)$, other assertions are verified as previously. Using the communication axiom $\mathbf{A}_{\mathcal{E}}$ **4**, the assignment rule $\mathbf{R}_{\mathcal{E}}$ **1** and the formation rule $\mathbf{R}_{\mathcal{E}}$ **16**, we get:

$$l_p = 1 \wedge \mathcal{P}_3(p) \wedge l_{p\times 2} = 3 \wedge m_i \neq M_i \wedge prod(m_i, M_i) \in \mathcal{M}_{RI} \wedge I$$
$$\{\ \mathrm{N}(2 \times \mathrm{p})\ ?\ \mathrm{y}_p;\ \mathrm{l}_p := \mathrm{l}_p + 1\ ||\ \mathrm{N(p)}\ !\ \mathrm{y}_{p\times 2} \times \mathrm{z}_{p\times 2}\ \}$$
$$l_p = 2 \wedge \mathcal{P}_3(p) \wedge l_{p\times 2} = 3 \wedge m_{p\times 2} \neq M_{p\times 2} \wedge prod(m_{p\times 2}, M_{p\times 2}) \in \mathcal{M}_{RI} \wedge I$$

Let us finally consider pairs of semantically matching communication commands within $N(p)$ and $N(p \times 2 + 1)$. We examine only the emission of $y_{p\times 2+1} \times z_{p\times 2+1}$ to $N(p)$, other assertions are checked as previously. Using the communication axiom $\mathbf{A}_{\mathcal{E}}$ **4**, the assignment rule $\mathbf{R}_{\mathcal{E}}$ **1** and the formation rule $\mathbf{R}_{\mathcal{E}}$ **16**, we have:

$$l_p = 2 \wedge \mathcal{P}_3(p) \wedge l_{p\times 2+1} = 3 \wedge m_i \neq M_i \wedge prod(m_i, M_i) \in \mathcal{M}_{ER} \wedge I$$
$$\{\ \mathrm{N}(2 \times \mathrm{p} + 1)\ ?\ z_p;\ \mathrm{l}_p := \mathrm{l}_p + 1\ ||\ \mathrm{N(p)}\ !\ \mathrm{y}_{p\times 2+1} \times \mathrm{z}_{p\times 2+1}\ \}$$
$$l_p = 3 \wedge \mathcal{P}_3(p) \wedge l_{p\times 2+1} = 3 \wedge m_{p\times 2+1} \neq M_{p\times 2+1} \wedge prod(m_{p\times 2+1}, M_{p\times 2+1}) \in \mathcal{M}_{RI} \wedge I$$

which concludes our proof □.

# 5 Implementation

We show here that every $\mathcal{E}_{CSP}$ program can be rewritten into a CSP program. We introduce a rewriting function $\mathcal{R}$ which is defined by induction on $\mathcal{E}_{CSP}$ commands. The proposed implementation is not concerned with efficiency, its purpose is mainly to show that our exception handling mechanism can be modelled through the use of CSP commands. For the sake of conciseness, we introduce only the rewriting of the most significant commands.

Signalling a global exception requires communications between the signaller and the processes catching the exception. This action is therefore implemented by means of send and receive commands interspersed in the text of destination processes. The places where receive commands should be inserted are exactly the places where exceptions can be caught. Let us recall that a process $P_i$ catches a global exception signalled by a process $P_j$ when $P_i$ communicates synchronously with $P_j$. Furthermore, exception catching has to be considered once a process is terminated in order to not introduce deadlock. For this purpose, the body $\mathcal{C}$ of any process is rewritten as the sequential composition of $\mathcal{R}(\mathcal{C})$ with a command $\mathcal{H}$ dedicated to the catching of global exceptions signalled after $\mathcal{C}$'s execution. These exceptions are not handled since the catching process is to be terminated. In the following, $\mathcal{R}(\mathcal{C})$ is called *process standard part* and $\mathcal{H}$ is called *process exceptional part*. The treatment of process exceptional termination is based on the work of [Apt et al.84] which shows that the distributed termination convention of the

CSP repetitive command can be explicitly modelled through the use of CSP commands.

We briefly summarize the proposal of [Apt et al.84]. Given a repetitive command $*[\Box_{j=1,\ldots,k}\ b_j;\ P_{i_j}\ \$_s\ x_j \rightarrow C_j]$ within a process $P_i$, it is rewritten as:

```
*[□j=1,...,k bj and continue(ij); Pij $s xj → Cj
  □j=1,...,k bj and continue(ij); Pij?s,end → continue(ij):= false]
```

where $continue(i_j)$ is initialized to true at the beginning of $P_i$. Furthermore, the following program section is inserted at the end of $P_i$:

```
*[□j:j∈Γi continue(j); Pj!s end → continue(j) := false
  □j∈Γi continue(j); Pj?s end → continue(j) := false ]
```

$\Gamma_i$ is the set of indices of all processes referred to within guards in some loop in $P_i$. $\Gamma_i$ can be statically determined. Process body termination modelling directly follows.

The rewriting of an $\mathcal{E}_{CSP}$ process into a CSP process uses sets which can be statically determined. Given an $\mathcal{E}_{CSP}$ process $P_i$, we define $\mathcal{G}_i$ to be the set of global exceptions signalled by $P_i$ and $\mathcal{P}_i$ to be the set of processes with which $P_i$ can potentially communicate. The following local declarations are inserted at the beginning of any rewritten process $P_i$:

```
type
  state = {exec, end, exc};
var
  cont: array [1..n] of state;
  exc_p: array [1..n] of string;
```

where

- the rewritten process is assumed to belong to a system of $n$ processes;
- $cont[j]$ indicates if $P_j$ is terminated or not;
- $exc\_p[j]$ denotes the global exception signalled by $P_j$ when $cont[j] = exc$.

Finally, all the elements of *cont* are initialized to *exec* at the beginning of $P_i$.

## 5.1 Command Rewriting

We examine the rewriting of main $\mathcal{E}_{CSP}$ commands. We assume that commands belong to a process $P_i$. A command of the form $C_1;\ C_2$ is rewritten as:

```
| R(C1);
| [ cont[i] = exc → skip □ not (cont[i] = exc) → R(C2) ]
```

The command **try** $C_1$ **except** $e_1$: $h_1$; ...; $e_m$: $h_m$ **end** is rewritten as the sequential composition of $\mathcal{R}(C_1)$ (1) with an alternative command (2). This last command is devoted to handling exceptions that may be signalled by $C_1$.

```
(1) | R(C1);

(2) | [ □j=1..m cont[i]=exc and exc_p[i] = "ej" → cont[i] := exec; R(hj);
    |   □ not (cont[i]=exc) → skip ];
```

When $P_i$ signals an exception $e$, *cont[i]* and *exc_p[i]* have to be updated. The additional actions to be performed when $e$ is global are carried out in the process exceptional part. $\mathcal{R}$(**signal** $e$) thus returns: *cont[i] := exc; exc_p[i] := "e";*.

The rewriting of "$P_j$ \$ $x$" introduces an alternative command. The execution of a communication command which names a process $P_j$ leads either to the execution of the corresponding CSP command (1) or to the catching of a global exception signalled by $P_j$. In this latter case, the exception may either be caught at the communication point (2) or may have been caught at an earlier communication (3).

```
[
(1) | not (cont[j] = exc); Pj $ x →
    |     skip
    |
    | □∀e∈G, not (cont[j] = exc); Pj ? e →
(2) |     cont[j] := exc; exc_p[j] := "e";
    |     cont[i] := exc; exc_p[i] := "e";
    |
    | □∀e∈G, cont[j] = exc →
(3) |     cont[j] := exc; exc_p[j] := "e";
    |     cont[i] := exc; exc_p[i] := "e";
]
```

This rewriting is not sufficient when the process sends the value of an expression; it should be checked for the possible occurrence of the predefined exception *unrep* while evaluating this expression. Due to the definition of $\mathcal{R}(P_j\$x)$, a deadlock in the original program now results in termination of the rewritten process since all the guards fail. Deadlock could be detected by means of rewriting. It would then be necessary to add the reception of the message *end* which is sent by $P_j$ and to replace the boolean expression **not** (*cont[j] = exc*) within guards (1) and (2) by the boolean expression *cont[j] = exec*.

We now define $\mathcal{R}(*[\Box_{k=1,\ldots,n} b_k; P_{i_k}\$x_k \rightarrow C_k])$ where original guarded commands appear in (1). Furthermore, the termination of process standard part has to be managed in any repetitive command. Additional guarded commands are therefore inserted (2). This last set of guarded commands follows the proposal of [Apt et al.84] about the explicit modelling of distributed termination convention of the CSP repetitive command. Finally, global exceptions may be caught when executing a repetitive command if communication commands appear within guards. This is handled by means of additional guarded commands (3).

```
*[
 (1)  | □k=1,...,n; cont[i] = exec and bk and cont[ik] = exec; Pik $ xk →
      |     R(Ck)
 (2)  | □k=1,...,n cont[i] =exec and bk and cont[ik] = exec; Pik ? end →
      |     cont[ik] := fin
      | □k=1,...,n,∀e∈Gk cont[i]=exec and bk and cont[ik] = exec; Pik ? e →
      |     cont[ik] := exc; exc_p[ik] := "e";
 (3)  |     cont[i] := exc; exc_p[i] := "e";
      | □k=1,...,n,∀e∈Gk cont[i]=exec and bk and cont[ik] = exc →
      |     cont[i] := exc; exc_p[i] := exc_p[ik];
];
```

Alternative commands are rewritten in the same way. Nonetheless, termination of the alternative command has then to be taken into account if none of its guards was selected, exception catching being perceived as a selection.

## 5.2 Process Exceptional Part

The *process exceptional part* serves two purposes: explicit management of process termination and (global) exception catching. A process $P_i$ does terminate once all the processes within the set $\mathcal{P}_i$ terminate. This can be expressed as: $\forall j, j \in \mathcal{P}_i : cont[j] \neq exec$. The exceptional part of $P_i$ is therefore written as:

```
*[
 □j,i∈P, not (cont[i]=exc) and cont[j]=exec; Pj!end → cont[j]:=end
 □j∈P, cont[j]=exec; Pj?end → cont[j]:=end
 □j,i∈P, cont[i]=exc and exc_p[i] ∈ Gi and cont[j]=exec; Pj!exc_p[i] → cont[j]:=end
 □j,i∈P, cont[i]=exc and exc_p[i] ∉ Gi and cont[j]=exec; Pj!end → cont[j]:=end
 □j∈P,,e∈G, cont[j]=exec; Pj?e → cont[j]:=end
];
```

# 6 Conclusions

We have proposed a model of exception handling for communicating sequential processes. Our main design goal was to favor the development of correct and robust parallel programs. This approach first led to extend the termination model defined for sequential programs. We then introduced two new notions to cope with parallelism: *global exceptions* and *exception catching*. These two notions enable simple management of the exceptional termination of a process. Notably, they solve the resulting problem of deadlocks. A *global exception* is an exception signalled by a process. Its occurrence indicates that its signalling process has not been able to ensure at least an expected cooperation with another process. A global exception is then *caught* by a process when the process attempts to communicate with the signalling process. To our knowledge, this precise definition of exception catching, essential for correctness proof of robust parallel programs, has only been addressed for parallel languages where processes communicate by remote operation call. This notion may be found in proposals for languages with a different communication model (e.g., [Szalas et al.85]) but an exception is then to be caught in *any* point of an executing process sub-action. As a result, the process state in which an exception is caught cannot be precisely defined. To demonstrate that our model of exception handling serves the design of correct robust parallel programs, we have presented a sound and relatively complete proof system for the CSP language enriched with our model of exception handling. To our knowledge, correctness proof of robust parallel programs has only been addressed in [Lodaya et al.90], where a proof system for a subset of ADA with exception handling is proposed. Finally, the adequacy of our model of exception handling with the underlying programming model has been shown. The programming language integrating the mechanism of exception handling expressing our model may be rewritten in terms of commands of the embedding language. The remainder of this section is devoted to a comparison of our proposal with related work.

Propagation of exceptions to remote processes has notably been addressed in [Szalas et al.85]. This investigates the extension of an exception handling mechanism expressing the continuation model. The host programming language offers dynamic creation of processes. An exception can be *explicitly* propagated from one process to another one through the use of a specific command. Synchronization commands define process fragments within which an exception signalled by another process may be caught. When an exception is caught, its handler is

sought according to the scheme defined in the sequential case. Exception propagation being explicit, propagation of exceptions for deadlock avoidance has to be managed by the programmer. Therefore, should an exception be caught by *more than one process, it must be explicitly signalled to each of those processes.* Besides, as we mentioned earlier, exception catching in any point of an executing process sub-action tends to prevent correctness proof of parallel programs in the presence of exceptions. The proposal of [Levin77] also introduces the notion of exception propagation to remote processes. It focuses on systems of processes communicating *via* object sharing. When an exception related to a shared object is signalled, all the modules sharing the object are made aware of the exception. *Within a module, the exception handler is sought along the invocation chain.* Handler activations depend upon the chosen policy among those offered by the mechanism. Termination and resumption facilities are offered and appropriate control structures are available to manage the parallel execution of handlers. The resumption facility is actually convenient for certain kinds of applications; a convincing example may be found in [Levin77]. However, in a parallel framework, this may somewhat be compared to the remote procedure call facility [Nelson81] whose integration in a programming language is to be preferred.

The *Ft-Actions* proposal of [Jalote et al.86] may be related to our work since it is also concerned with the introduction of an exception handling facility within CSP. An *Ft-Action* defines a fault tolerant parallel sub-action within a CSP program. *Ft-actions* can be statically nested. As we mentioned earlier, we believe that the facility of nested parallel operations should be provided independently of an exception handling mechanism. Furthermore, correctness issues are not addressed in [Campbell et al.86] even though rewriting of *FT-Actions* in CSP may be advocated. However, as the rewriting introduces a centralized manager which relies heavily on the use of communication commands within guards, the proof system of [Apt83], proven sound and complete, cannot be used to prove partial correctness of rewritten programs. Furthermore, an exception may be caught at any point of execution. Hence, the setting of precise properties about processes in the presence of exceptions is somewhat prevented. Moreover, a process catching an exception terminates. It follows that the proposed model of exception handling can hardly be retained in absence of nesting of parallel operation. Finally, let us indicate that this proposal expresses the model of exception handling introduced in [Campbell et al.86], a model which has also been used in [Taylor86].

Other proposals for imperative programming languages with explicit parallelism include those of [Ada83, Liskov et al87, Strom et al.83, Ichisugi et al.90]

and [Huang et al.90]. These contributions are mainly concerned with exception handling in the presence of synchronous or asynchronous remote operation call. In the ADA mechanism [Ada83], any attempt to communicate with a task which is terminated, exceptionally or not, leads to signal the predefined exception *tasking_error* to the requesting task. In our solution, a process which attempts to communicate with an exceptionally terminated process catches a global exception instead of a predefined exception. The catching process may then grasp conditions under which the signaller was terminated. Such information could actually be exploited in the ADA language. However, this would require to explicitly program an *exception server*, which would register termination condition of all the exceptionally terminated tasks. This server would then be consulted by tasks handling the exception *tasking_error*.

The fact that our design choices were guided by correctness issues enabled us to define a rigorous and concise model of exception handling. We have been able to extend the model in a straightforward way to cope with nesting of parallel blocks and a communication model akin to the *multiway rendezvous*. The obtained model has also been integrated in a programming language for which we provided a proof system [Issarny91a]. We were then able to extend this model in order to get a general model of exception handling for systems of communicating processes [Issarny91b]. This general model has been integrated in a parallel object oriented language with dynamic creation of processes and asynchronous communication [Benveniste et al.92].

ACKNOWLEDGMENTS

The authors wish to express their gratitude to P. Le Guernic for helpful discussions on the subject of this paper. They also would like to thank H. M. Levy for his useful comments on an earlier version of this paper.

# References

[Ada83] Ada. *The Programming Language ADA*. Lecture Notes in Computer Science 155, 1983.

[Apt et al.80] Apt (K. R.), Francez (N.) and de Roever (W. P.). A proof system for communicating sequential processes. *ACM transactions on programming languages and systems*, vol. 2 (3), July 1980, pp. 359–385.

[Apt et al.84] Apt (K. R.) and Francez (N.). - Modeling the distributed termination convention of CSP. *ACM transactions on programming languages and systems*, vol. 6 (3), July 1984, pp. 370–379.

[Apt et al.91] Apt (K. R.) and Olderog (E. R.). - *Verification of Sequential and Concurrent Programs*. - Springer-Verlag, 1991, *Texts and Monographs in Computer Science*.

[Apt83] Apt (K. R.). - Formal justification of a proof system for communicating sequential processes. *Journal of the ACM*, vol. 30 (1), January 1983, pp. 197–216.

[Bal et al.89] Bal (H.), Steiner (J.) and Tanenbaum (A.). - Programming languages for distributed computing systems. *ACM computing surveys*, vol. 21 (3), September 1989, pp. 261–322.

[Banatre et al.86] Banâtre (J. P.), Banâtre (M.) and Ployette (F.). - The concept of multi-functions, a general structuring tool for distributed operating system. *In : Proceedings of the 6th Distributed Computing Systems Conference*. - Cambridge, MA, May 1986.

[Banatre90] Banâtre (J. P.). - *La programmation parallèle : outils, méthodes et éléments de mise en œuvre*. - Eyrolles, 1990.

[Benveniste et al.92] Benveniste (M.) and Issarny (V.). - ARCHE *: un langage parallèle à objets fortement typé*. - Research Report 642, Rennes, France, IRISA, March 1992.

[Campbell et al.86] Campbell (R. H.) and Randell (B.). - Error recovery in asynchronous systems. *IEEE transactions on software engineering*, vol. SE-12 (8), August 1986, pp. 811–826.

[Cocco et al.82] Cocco (N.) and Dully (S.). - A mechanism for exception handling and its verification rules. *Computer language*. vol. 7, March 1982, pp. 43–49.

[Cristian82] Cristian (F.). - Robust data types. *Acta Informatica*, vol. 7 (9), October 1982, pp. 365–397.

[Cristian84] Cristian (F.). – Correct and robust programs. *IEEE transactions on software engineering*, vol. SE-10 (2), March 1984, pp. 163–174.

[Dijkstra75] Dijkstra (E. W.). – Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, vol. 18 (8), August 1975, pp. 453–457.

[Francez et al.86] Francez (N.), Hailpern (B.) and Taubenfeld (G.). – SCRIPT: a communication abstraction mechanism and its verification. *Science of Computer Programming*, vol. 6, 1986, pp. 35–88.

[Goodenough75] Goodenough (J. B.). – Exception handling issues and a proposed notation. *Communications of the ACM*, vol. 18 (12), December 1975, pp. 683–696.

[Hoare69] Hoare (C. A. R.). – An axiomatic basis for computer programming. *Communications of the ACM*, vol. 12, 1969, pp. 576–580.

[Hoare78] Hoare (C. A. R.). – Communicating sequential processes. *Communications of the ACM*, vol. 21 (8), August 1978, pp. 666 674.

[Huang et al.90] Huang (D. T.) and Olsson (R. A.). – An exception handling mechanism for SR. *Computer language*, vol. 15 (3), 1990, pp. 163–176.

[Ichisugi et al.90] Ichisugi (Y.) and Yonezawa (A.). – Exception handling and real time features in an object-oriented concurrent language. *In : Concurrency: theory, language, and architecture. UK/Japan workshop*, LNCS 491. pp. 604–615. Springer-Verlag.

[Issarny90] Issarny (V.). – Design and implementation of an exception handling mechanism for communicating sequential processes. *In : Proc. of CONPAR 90 - VAPP IV, Joint Conference on Vector and Parallel Processing, LNCS 457*, pp. 604–615. Zurich, Switzerland, September 1990.

[Issarny91a] Issarny (V.). An exception handling model for parallel programming and its verification. *In : Proc. of the ACM SIGSOFT'91 Conference on Software for Critical Systems*, pp. 92-100. - New Orleans, La, December 1991.

[Issarny91b] Issarny (V.). *Un modèle pour le traitement des exceptions dans les programmes parallèles.* - Thèse de doctorat, Université de Rennes I, November 1991.

[Jalote et al.86] Jalote (P.) and Campbell (R. H.). - Atomic actions for fault tolerance using CSP. *IEEE transactions on software engineering*, vol. SE-12 (1), January 1986, pp. 59–68.

[Knudsen87] Knudsen (J. L.). - Better exception handling in block structured systems. *IEEE Software*, vol. 17 (2), May 1987, pp. 40–49.

[Levin77] Levin (R.). - *Program structures for exceptional condition handling.* - PhD thesis, Carnegie-Mellon University, June 1977.

[Liskov et al.79] Liskov (B. H.) and Snyder (A.). - Exception handling in Clu. *IEEE transactions on software engineering*, vol. SE-5 (6), November 1979, pp. 546–558.

[Liskov et al87] Liskov et al. - *Argus Reference Manual.* - Research report MIT/LCS/TR-400, MIT. Laboratory for Computer Science, 1987.

[Lodaya et al.90] Lodaya (K.) and Shyamasundar (R. K.). - Proof theory for exception handling in a tasking environment. *Acta Informatica*, vol. 28, 1990, pp. 365–397.

[Luckham et al.80] Luckham (D. C.) and Polak (W.). - ADA exception handling: an axiomatic approach. *ACM transactions on programming languages and systems*, vol. 2 (2), April 1980, pp. 225-233.

[Meyer88] Meyer (B.). *Object-oriented software construction.* Prentice-Hall International, 1988.

[Nelson81] Nelson (B. J.). - *Remote Procedure Call.* - PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, 1981.

[Owicki et al.76] Owicki (S.) and Gries (D.). - A consistent and complete deductive system for verification of parallel programs. *In : Proceedings of the 8th annual ACM symposium on Theory of computing*, pp. 73–86. - Hershey, Pa, May 1976.

[Rovner et al.85] Rovner (P.), Levin (R.) and Wick (J.). - *On extending MODULA-2 for building large, integrated systems.* - Technical report, Palo Alto, Ca, Digital Systems Research Center, January 1985.

[Strom et al.83] Strom (R. E.) and Yemini (S.). - NIL: An integrated language and system for distributed programming. *ACM SIPLAN Notices*, vol. 18 (6), June 1983, pp. 73–82.

[Szalas et al.85] Szalas (A.) and Szczepanska (D.). - Exception handling in parallel computations. *ACM SIGPLAN notices*, vol. 20 (10). October 1985, pp. 95–104.

[Szczepanska91] Szczepanska (D.). - A Hoare-like verification system for a language with an exception handling mechanism. *Theorical computer science*, vol. 80, 1991, pp. 319–335.

[Taylor86] Taylor (D. J.). - Concurrency and forward error recovery in atomic actions. *IEEE transactions on software engineering.* vol. SE-12 (1), January 1986, pp. 69–78.

[Yemini et al.85] Yemini (S.) and Berry (D. M.). - A modular verifiable exception handling mechanism. *ACM transactions on programming languages and systems*, vol. 7 (2), April 1985, pp. 214–243.

[Yemini et al.87] Yemini (S.) and Berry (D. M.). - An axiomatic treatment of exception handling in an expression oriented language. *ACM transactions on programming languages and systems*, vol. 9 (3), July 1987, pp. 390–407.

## LISTE DES DERNIERES PUBLICATIONS INTERNES PARUES A L'IRISA

PI 650 BLOCK-ARNOLDI AND DAVIDSON METHODS FOR UNSYMMETRIC LARGE EIGENVALUE PROBLEMS
Miloud SADKANE
Avril 1992, 24 pages.

PI 651 COMPILING SEQUENTIAL PROGRAMS FOR DISTRIBUTED MEMORY PARALLEL COMPUTERS WITH PANDORE II
Françoise ANDRE, Olivier CHERON, Jean-Louis PAZAT
Avril 1992, 18 pages.

PI 652 CHARACTERIZING THE BEHAVIOR OF SPARSE ALGORITHMS ON CACHES
Olivier TEMAM, William JALBY
Avril 1992, 20 pages.

PI 653 MADMACS : UN OUTIL DE PLACEMENT ET ROUTAGE POUR LE DESSIN DE MASQUES DE RESEAUX REGULIERS
Eric GAUTRIN, Laurent PERRAUDEAU, Oumarou SIE
Avril 1992, 16 pages.

PI 656 CAUSALITY ORIENTED SHARED MEMORY FOR DISTRIBUTED SYSTEMS
Michel RAYNAL, Masaaki MIZUNO, Mitch NEILSEN
Avril 1992, 8 pages.

PI 657 ALGORITHMES PARALLELES POUR L'ANALYSE D'IMAGE PAR CHAMPS MARKOVIENS
Etienne MEMIN, Fabrice HEITZ
Mai 1992, 74 pages.

PI 658 MULTISCALE SIGNAL PROCESSING : ISOTROPIC RANDOM FIELDS ON HOMOGENEOUS TREES
Bernhard CLAUS, Ghislaine CHARTIER
Mai 1992, 28 pages.

PI 659 ON THE COVARIANCE-SEQUENCE OF AR-PROCESSES. AN INTERPOLATION PROBLEM AND ITS EXTENSION TO MULTISCALE AR-PROCESSES
Bernhard CLAUS, Albert BENVENISTE
Mai 1992, 36 pages.

PI 660 EXCEPTION HANDLING IN COMMUNICATING SEQUENTIAL PROCESSES DESIGN, VERIFICATION AND IMPLEMENTATION
Jean-Pierre BANATRE, Valérie ISSARNY
Mai 1992, 38 pages.



Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249-6399